Migrations Guidelines

# Problemática a resolver:

Actualmente existen varias formas de implementar la estrategia, unas no son muy eficientes, otras pudieran requerir cambios grandes y complejos, principalmente en los consumidores de dichos componentes.

La presente propuesta, pretende establecer una estrategia estándar de trabajo que balance la eficiencia, reduzca los cambios necesarios y nos provea una experiencia de desarrollo eficiente para futuros componentes.

# Inputs:

(One-way data binding)

## Transformación de versión anterior a versión actual:

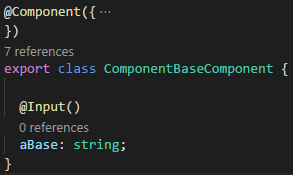
(ilustrado)

### Versión Actual:

(simplificado)

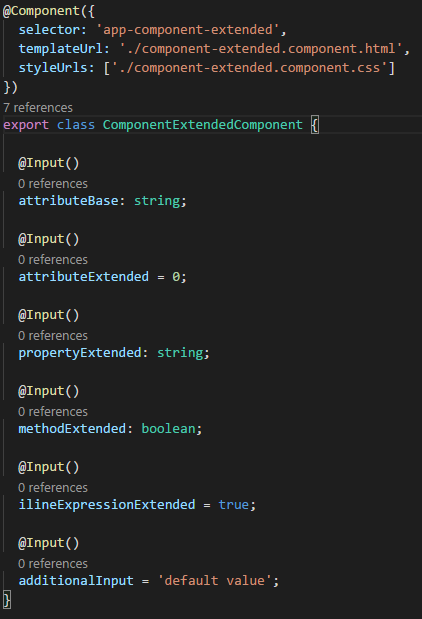
#### Component:

##### Component Base:



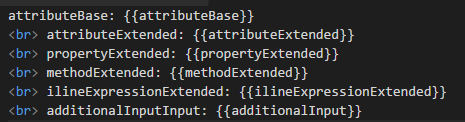
* Viene de producto y posee los inputs definidos ahí.
* Si nosotros necesitábamos agregar nueva funcionalidad, extendíamos el componente:

##### Component Extended:



* Debíamos incluir todos los inputs del componente base y definir adicionalmente los específicos a nuestra lógica extendida.
* **Notar que esto es una versión MUY simplificada del componente extendido con el único propósito de establecer el punto de partida.**

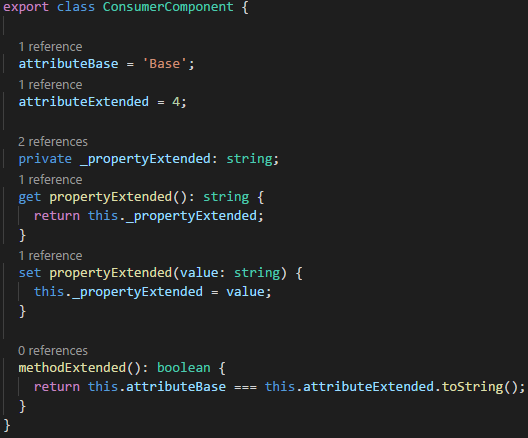
##### Template:



* En el caso del template, siempre es el mismo y sobrescribimos la versión que nos llegaba de producto.
* El ejemplo muestra un fragmento de la plantilla en el que se consumen los inputs del componente.

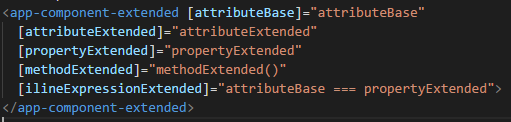
#### Consumer:

##### Consumer Component:

A la hora de consumir dicho componente, bastaba con bindearle los inputs que necesitábamos en cada caso.  


* Componente **consumer** de muestra.
* Posee datos para bindear tanto al componente base como a los inputs del extendido.
* Define los tipos principales de bindeo que puede realizar: atributo, propiedad y método.

##### Template:

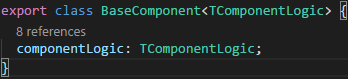


* Fragmento del template del **consumer** en el que utiliza el **componente.**
* Provee los bindings que define en el componente.
* Provee un binding a modo de expression.
* No provee binding para el input **additionalInput**, por lo que el mismo mantendrá su valor por defecto asignado en el **componente**.

### Versión Migrada:

En la nueva versión se establece una ruptura entre la definición de un componente y su lógica, dividiendo el anterior componente en 2 clases:

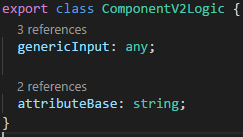
#### Component:

  
**Versión simplificada del BaseComponent<T>**



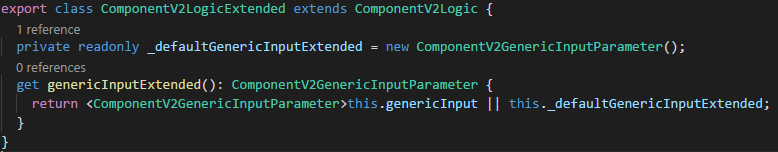
* El nuevo componente tendrá ahora el único objetivo de definir el único componente (su selector, template, estilos, input, output, etc…)
* No poseerá lógica ni atributos, simplemente sería un wrapper alrededor de la clase de la lógica, la cual sería la que se extendiese.
* Para manejar la aparición de nuevos inputs, se define un **@Input() genericInput: any**. El cual se orienta para ser utilizado en TODOS los casos de que se precise un nuevo input.
* En su constructor, éste crea una referencia a su lógica, la cuál sería utilizada para acceder tanto a sus atributos como a sus métodos.
* Extiende una clase genérica con el objetivo de proveer un chequeo de tipos a la hora de extender la lógica.

##### Logic:



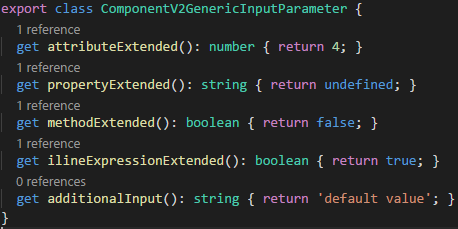
* La lógica también es proveída por Producto y es en donde se almacena toda la información y la lógica asociada al componente.
* En este caso es la que posee el @Input que en la versión anterior pertenecía al ComponentBase (attributeBase)
* También posee el atributo del genericInput.
* A la hora de extender el componente, en esta nueva versión extenderíamos únicamente la lógica, incluyendo los inputs adicionales.
* La idea para facilitar el trabajo con los nuevos inputs (sólo los que precisa nuestro anterior componente extendido) sería crear una clase donde cada una de sus propiedades sea los inputs mencionados (profundizamos más adelante)

##### Logic Extended:



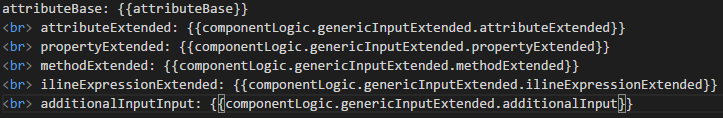
* El punto clave acá, es que debemos crear una propiedad para castear el genericInput: any, a la clase que nosotros queremos que represente nuestros inputs.
* Además, debemos crear una instancia por defecto para asegurarnos del caso en el que ningún **customer** proporcione un **generic input** y poder acceder a los valores por defecto definidos.

##### GenericInputParameter:



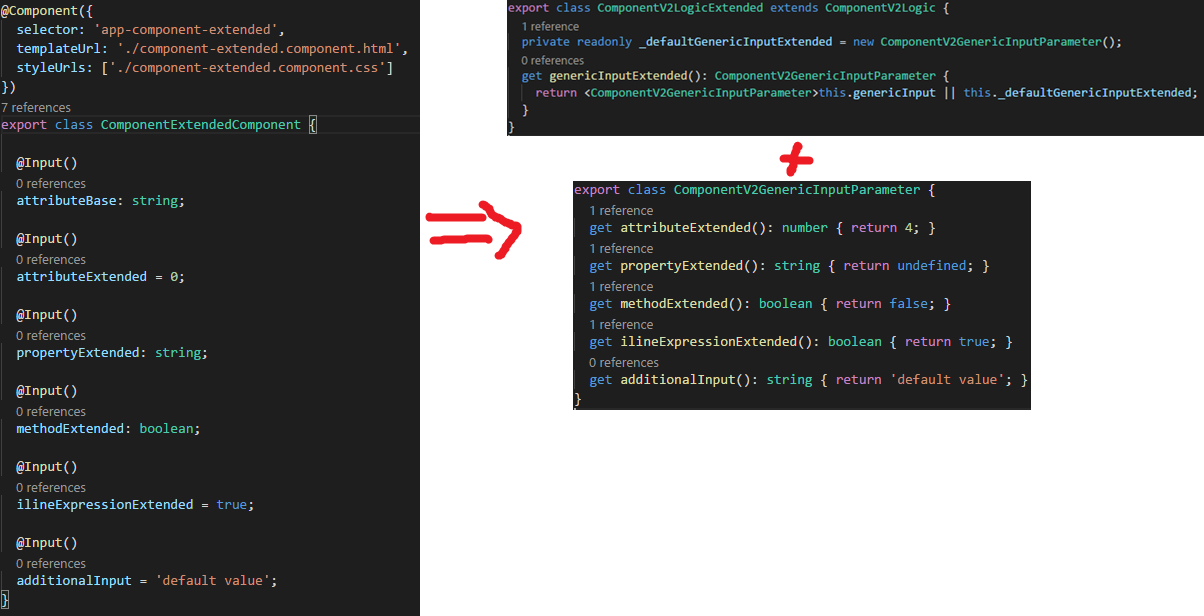
* Cada lógica extendida, debería definir una clase **genericInputParameter** la cual incluye una propiedad de sólo lectura (get) por cada input adicional y define el valor por defecto de cada Input (sea undefined, null, false, etc…)
* Como nuestra lógica extendida ahora dependerá de nuestra clase **genericInputParameter** podemos acceder al valor de dichos atributos a través del  **genericInputParameter**.

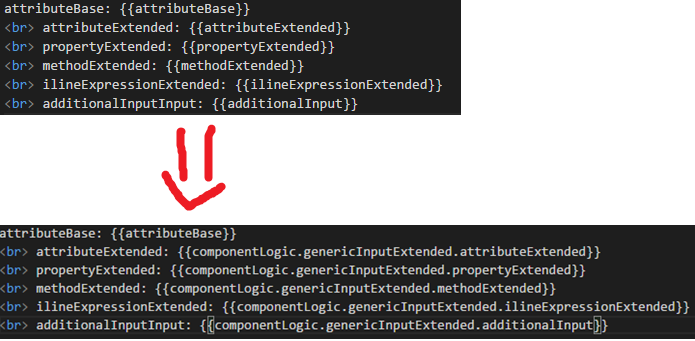
##### Template:



* Finalmente, sólo debemos actualizar el template para referirnos a los atributos **a tráves** del nuevo **genericInputExtended**.

##### Component Extended (previous) Vs. LogicExtended (current):





#### Consumer:

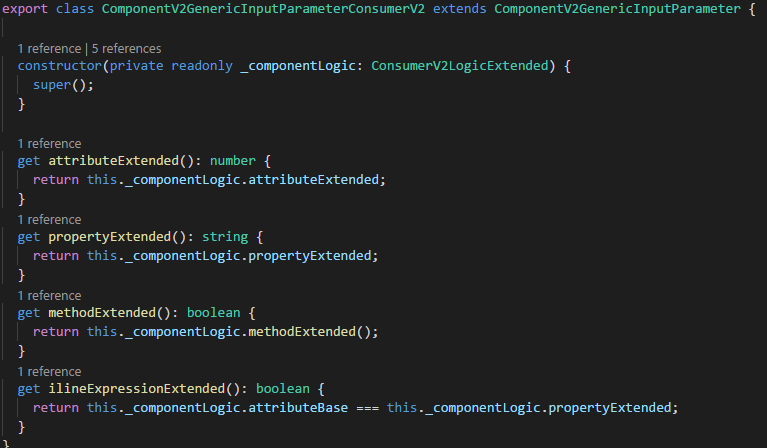
* Debemos notar que el consumer también sería migrado y tendría su componente separado de su logic.
* Si en un **consumer** anterior utilizábamos los inputs del **component** extendido, indudablemente dicho **consumer** también era un componente extendido, por lo tanto **CUALQUIER** modificación que requiera la aplicación de esta estrategia se haría **ÚNICAMENTE** en la lógica extendida y **NUNCA** en el producto.

##### Logic Extended:



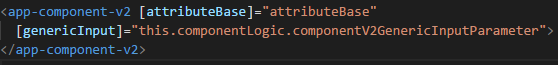
* La lógica extendida del **consumer** sería como se esperaría. Sin ningún cambio aparente en su estructura anterior (sin considerar los inputs nuevos).
* El único cambio es que crearíamos una especialización del **generic input parameter** que provee el **component** la cual instanciamos **pasándole una referencia al logic extended del consumer**:

##### Generic Input Parameter:



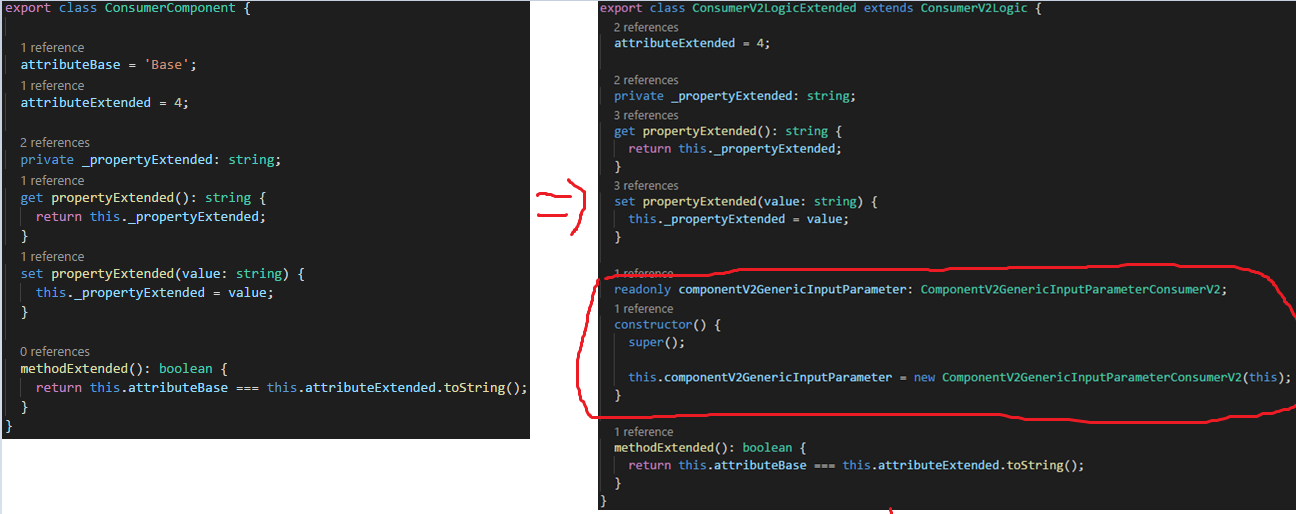
* En dicha clase, únicamente sobrescribiríamos las propiedades que el **consumer** utilizaba como inputs. En el ejemplo no se sobreescribe la propiedad additionalInput por lo que el **component** recibiría siempre el valor por defecto.
* La implementación de cada una de las propiedades sería lo mismo que antes incluíamos en el template del **consumer** al bindear con los inputs del **component** extendido.
* Cada implementación, utilizaría el **\_componentLogic** como contexto.
* Clave notar **el constructor** de la clase, el cual define y recibe una referencia al **\_componentLogic** que lo crea, en este caso la lógica extendida del **consumer.**

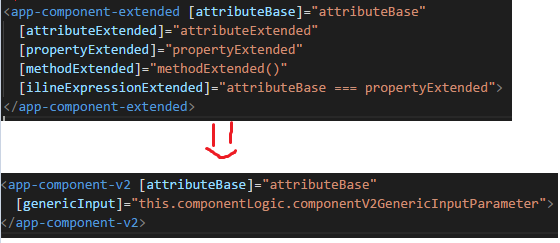
##### Template:



* El template del **consumer** reemplazaría el bindeo de cada uno de los **inputs** del **component** extendido por el único **genericInput** bindeándole entonces el instanciado **generic input parameter**.

##### Consumer (previous) Vs. Consumer (current)





## Pasos para seguir:

(a modo de resumen)

### Migración del Componente:

1. Crear la clase que representará al **generic input parameter**, la misma poseerá un getter por cada input propio del **logic extended** del **component**.
2. **XXXXXXXX DÓNDE PONERLA XXXXXXXXXX**
3. En la **logic extended** del **component**, crear un **get genericInputExtended** con el objetivo de castear el **genericInput: any** de su clase base al tipo de **generic input parameter** creado.
4. Además, crear un atributo privado readonly que guardará el valor por defecto del generic input para el caso en que ningún **consumer** lo proporcione.
5. Actualizar el template del **component** para que todas las referencias a los **inputs** adicionales se resuelvan mediante **componentLogic.genericInputExtended**.

### Consumers:

1. Notar que los consumidores del **component** no podrán acceder a los **input** adicionales directamente.
2. Crear la clase que representará al **generic input parameter** del **component** desde el punto de vista del **consumer**. La misma extendería la clase abstracta proveída por el **component** y **únicamente** **sobrescribiría** los getters que correspondan a los inputs que consume.
3. **XXXXXXXX DÓNDE PONERLA XXXXXXXXXX**
4. La clase anteriormente descrita además tendrá un atributo privado y de sólo lectura que representará a la **logic extended** del **consumer**. Dicho atributo será inyectado en su constructor.
5. Agregar a la **logic extended** del **consumer** un atributo privado de sólo lectura que representará el **generic input parameter** creado en los puntos anteriores. El mismo será instanciado en el constructor proporcionándole una referencia a **this**.
6. Reemplazar en el template del **consumer**, todos los bindeos a los distintos inputs, y bindear únicamente en **generic input parameter** creado en los pasos anteriores.

## Funcionalidades:

### Consumer:

* Bindear un valor, atributo o propiedad.
* Bindear un método.
* Bindear una expresión (inline)

### Component:

* Valor por defecto.
* ~~Valor inicial.~~

## Ventajas y Deficiencias Actuales:

* (V) El **consumer** no requiere especificar todos los parámetros que espera recibir el **component**.
* (V) Técnicamente no se establece un one-way binding, por lo que el performance mejora algo.
* (V) El **generic parameter** no es intrusivo, osea, que no es necesario alterar el contenido de los **consumers** para aplicar la versión actual.
* (V) Los cambios en el **component** son mínimos para aplicar la versión actual.
* (D) No existe un valor inicial. Es posible asignarse un valor por defecto pero el mismo depende del valor proporcionado (null / undefined) pero no es posible establecer un valor inicial que será modificado en cuanto el **consumer** genere un valor.
* (D) El **component** no puede cambiar el valor del input. Desde el punto de vista del **component** el valor de su **input** sería de sólo lectura.

# Outputs:

(event binding)

## Transformación de versión anterior a versión actual:

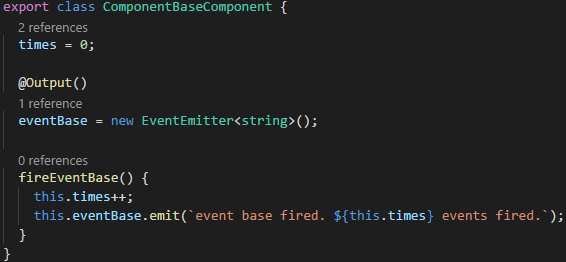
(ilustrado)

### Versión Actual:

(simplificado)

#### Component:

##### Component Base:



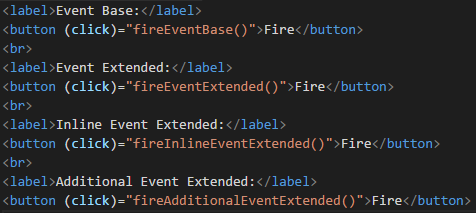
* Viene de producto y posee los outputs definidos ahí.
* Si nosotros necesitábamos agregar nueva funcionalidad, extendíamos el componente:

##### Component Extended:



* Debíamos incluir todos los outputs del componente base y definir adicionalmente los específicos a nuestra lógica extendida.
* Típicamente, definíamos nuestros outputs (de tipo EventEmitter<T>) y los disparadores de los mismos.
* **Notar que esto es una versión MUY simplificada del componente extendido con el único propósito de establecer el punto de partida. Se repiten algunos atributos pues, en este ejemplo, no hay ninguna herencia.**

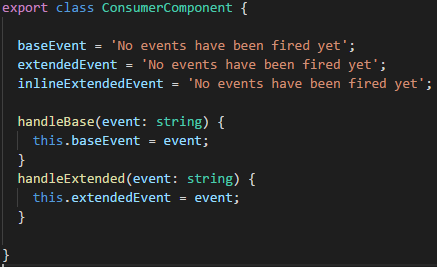
##### Template:



* En el caso del template, siempre es el mismo y sobrescribimos la versión que nos llegaba de producto.
* El ejemplo muestra un fragmento de la plantilla en el que se disparan los eventos del componente.

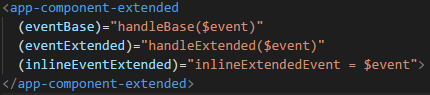
#### Consumer:

##### Consumer Component:

A la hora de consumir dicho componente, bastaba con bindearle un manejador a los outputs que necesitábamos en cada caso.  


* Componente **consumer** de muestra.
* Posee los métodos que manejarían cada evento del **component**.

##### Template:

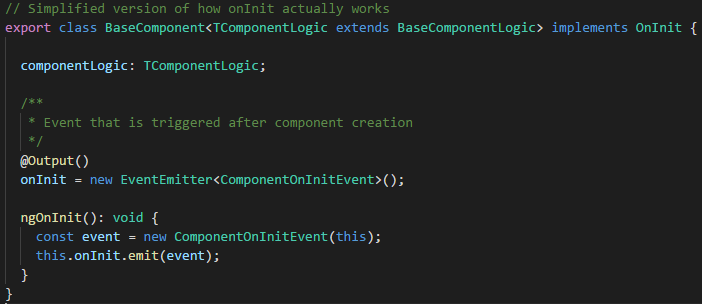


* Fragmento del template del **consumer** en el que utiliza el **componente.**
* Provee los bindings que define en el componente.
* Provee un binding a modo de expression.
* No provee binding para el output **additionalEvent**, por lo que nunca será notificado de dicho evento.

### Versión Migrada:

En la nueva versión se establece una ruptura entre la definición de un componente y su lógica, dividiendo el anterior componente en 2 clases:

#### Component:

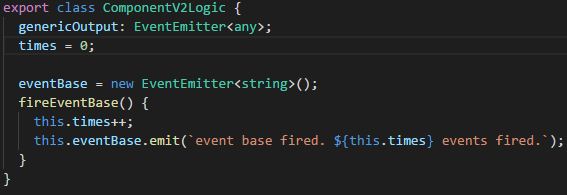
  
**Versión simplificada del BaseComponent<T>**

* **Para el futuro,** notar que el BaseComponent define un **@Output onInit**. Dicho output permite al **consumer** subscribirse al evento y el tipo **ComponentOnInitEvent** proveerá una referencia al **component** que genera dicho evento.



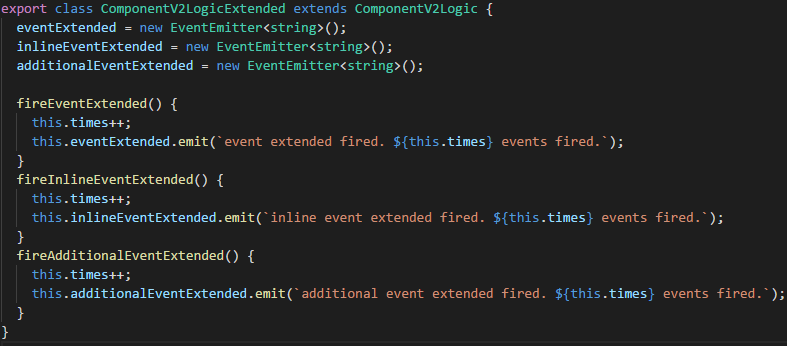
* El nuevo componente tendrá ahora el único objetivo de definir el único componente (su selector, template, estilos, input, output, etc…)
* No poseerá lógica ni atributos, simplemente sería un wrapper alrededor de la clase de la lógica, la cual sería la que se extendiese.
* Para manejar la aparición de nuevos outputs, se define un **@Output() genericOutput: EventEmitter<any>**. El cual se orienta para ser utilizado en TODOS los casos de que se precise un nuevo input.
* **[IMPORTANTE]** La estrategia que se propone en este documento **NO** utiliza dicho **genericOutput**, se incluye en el manual por motivos de completitud.
* En su constructor, éste crea una referencia a su lógica, la cuál sería utilizada para acceder tanto a sus atributos como a sus métodos.
* Extiende una clase genérica con el objetivo de proveer un chequeo de tipos a la hora de extender la lógica.

##### Logic:



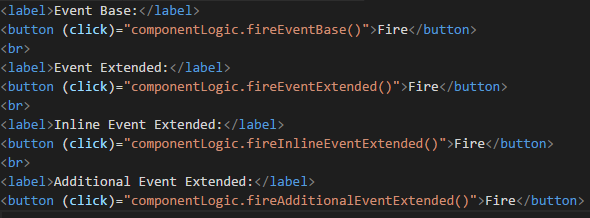
* La lógica también es proveída por Producto y es en donde se almacena toda la información y la lógica asociada al componente.
* En este caso es la que posee el @Output que en la versión anterior pertenecía al ComponentBase (attributeBase)
* También posee el atributo del genericOutput.
* A la hora de extender el componente, en esta nueva versión extenderíamos únicamente la lógica, incluyendo los outputs adicionales.
* En la estrategia que se propone **NO SE MODIFICARÍA** el componente en lo absoluto. El único cambio sería mínimo y a realizar **ÚNICAMENTE** en el **consumer** a la hora de subscribirse a los outputs.

##### Logic Extended:



* Como ven, salvo de la migración esperada, el **LogicExtended** del **component** **NO** posee ninguna adición especial para proveer sus propios **outputs**.

##### Template:



* Únicamente se actualiza la template para acceder a los métodos específicos de la **LogicExtended** como hiciéramos en cualquier otro caso.

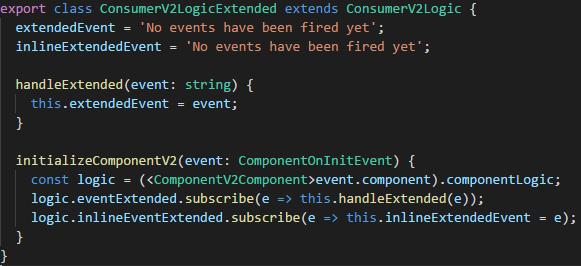
##### Component Extended (previous) Vs. LogicExtended (current):

**No existe ningún cambio específico entre ambas versiones.**

#### Consumer:

* Debemos notar que el consumer también sería migrado y tendría su componente separado de su logic.
* Si en un **consumer** anterior utilizábamos los outputs del **component** extendido, indudablemente dicho **consumer** también era un componente extendido, por lo tanto **CUALQUIER** modificación que requiera la aplicación de esta estrategia se haría **ÚNICAMENTE** en la lógica extendida y **NUNCA** en el producto.

##### Logic Extended:



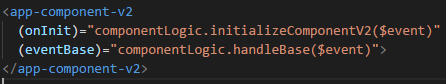
* La lógica extendida del **consumer** sería como se esperaría. Sin ningún cambio aparente en su estructura anterior (sin considerar los outputs nuevos).
* El único cambio es que nos registraríamos a los eventos del **component** extendido de forma programática, osea, en la **LogicExtended** del **consumer**, en lugar de en la **template** como haríamos anteriormente.
* Para realizar esto, nos aprovechamos del **ComponentOnInitEvent** que emite el **component** en su evento **(onInit)**, de esta forma, podemos obtener una referencia directa a la **LogicExtended** del **component**.
* Ya con la referencia de la **lógica del componente hijo**, nos subscribimos directamente a cada uno de sus eventos **como mismo haríamos en la template.**
* **[IMPORTANTE]** Por detrás, cuando Ng procesa los outputs establece la subscripción por nosotros, pasándole la expresión incluída en el template como parámetro del **subscribe**. Notar que, al realizarlo de forma manual, debe ser con el siguiente formato:

  
**VS**  


* En el caso de las inline expressions, quedan justo como estaban definidas en el template:

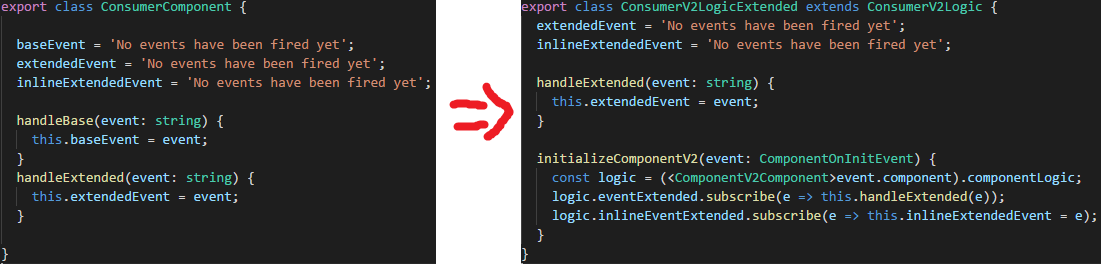
  
**VS**  

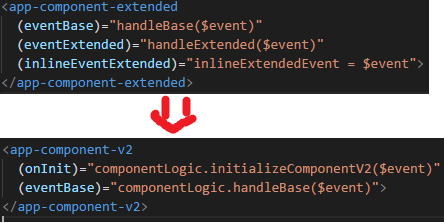

##### Template:



* El template del **consumer** reemplazaría el bindeo de cada uno de los **outputs** del **component** extendido por una subscripción al **(onInit)** del **component**..

##### Consumer (previous) Vs. Consumer (current)





## Pasos para seguir:

(a modo de resumen)

### Migración del Componente:

**El componente no requiere ninguna consideración especial al migrar outputs.**

### Consumers:

1. Notar que los consumidores del **component** no podrán acceder a los **outputs** adicionales directamente.
2. En la **logic extended** crear el método que se encargará de subscribirse a los **outputs** adicionales.
3. En el **template** subscribirse al evento **(onInit)** del componente en cuestión especificando el método creado anteriormente.
4. Utilizar el parámetro emitido por el evento **(onInit)** para obtener la **logicExtended** del componente.
5. Subscribirnos a los eventos que representan cada uno de los **outputs** adicionales.

# ViewChilds:

La estrategia para migrar los ViewChilds es bastante sencilla si construimos en una idea clave utilizada al migrar los outputs:

**Podemos subscribirnos al evento (onInit) de un componente para obtener una referencia al mismo.**

De esta forma, basta con replicar los pasos utilizados para migrar un generic input hasta el punto de obtener una referencia la componente y asignarle dicha referencia a nuestro **ViewChild:**



